Dynamic Shapes: A brief survey

TL;DR: We want to be able to extract graphs from PyTorch programs and re-use these graphs with input Tensors of varying shapes.
Status: We have designs for Python and C++. Nick Korovaiko is executing on the design for C++.

I. What do we want to do?

The most general formulation of the problem is: given a function (or PyTorch program), we would like to extract a graph via tracing that represents the function regardless of the shapes of input Tensors. The graph may be captured by LazyTensor or AOTAutograd and may be lowered to backends like XLA.

For example, given the following function:

```
def sin_then_flatten(x):
  y = x.sin()
  numel = y.shape[0] * y.shape[1]
  return y.view(numel)
```

we would like to extract a graph that looks like the following:

```
y = x.sin()
yshape0 = y.shape[0]
yshape1 = y.shape[1]
numel = yshape0 * yshape1
z = y.view(numel)
```

There are a number of potential things that we'd like to do once we have a graph:

- **Send it to a backend compiler.** Today, torch/xla traces out and recompiles a different graph every time input shapes change. Constant recompilation is wildly inefficient for models that use a lot of tensors that have different shapes like Mask RCNN and language models where input sentences have different word lengths. (see this note from FAIR researchers for empirical observations). If we have a graph that works for "all shapes" (or most of them), then we can decrease the number of recompilations.
- **Use it as a deployment solution**: deploy the graph (or a number of them) instead of the original Python code. For models that use tensors with different shapes, deploying the model would require deploying a different graph per input shape used -- this becomes impractical as the number of shapes goes up.

Furthermore, although this doc is titled "Dynamic Shapes", what we would really like is for the graph to work regardless of other properties of the input Tensors (like strides, contiguity, values).

II. Why can’t we do it?

Existing tracing mechanisms bake shapes/properties into the graph

```
def sin_then_flatten(x):
  y = x.sin()
  numel = y.shape[0] * y.shape[1]
  return y.view(numel)
```

If we were to use LazyTensor or AOTAutograd to trace the above code in PyTorch today, it would first require the shapes of the input (let’s say [2, 3]) and then produce a graph like the following:

```
y = x.sin()
z = y.view(6)
```

This is kind of a problem -- the captured graph only works for inputs that are viewable as shape (6,)!

The reason why this happens is that y.shape returns a Python Number, numel is computed using Python math, and then finally PyTorch sees a call to “y.view(6)”.

Shape assertions aren’t captured in a graph

If x.shape returns a Tuple[int, …], and y.shape returns a Tuple[int, …], the following assertion happens in Python. Since none of it goes through the PyTorch dispatcher, neither AOTAutograd nor LazyTensor see it...

```
def my_mm(x, y):
  assert x.shape[1] == y.shape[0]
  return ...
```

This happens in both user-level code and framework-level code

As we saw above, user code (written in Python) can bake in shapes. The same can happen to PyTorch code that is written in C++. Attempting to trace code that calls torch.flatten (implementation roughly below) with LazyTensor or AOTAutograd will also bake numel into the graph.

```
Tensor flatten(const Tensor& x) {
```

```
  int64_t numel = c10::multiply_ints(x.sizes().begin(), x.sizes().end());
  return x.view(numel)
}
```

Finally, there are places in subsystem code (e.g. Autograd) that bake in sizes and other properties. For example, the autograd formula for Tensor.expand saves the input sizes for backwards:

```
- name: expand(Tensor(a) self, int[] size, ...) -> Tensor(a)
  self: at::sum_to(grad, self.sizes())
```

Actionable:

- Some analysis of what all the problems are would give more flavor to the problem. E.g. Dynamic shape exploration.
- Analysis of uses of size-related methods in maskrcnn. Sheet 2 contains broader categories that still need to be drilled into.
- Analysis of uses of size-related methods in Bert

III. What are some approaches to solving this?

Claim: Rank-specialization is OK

It's really difficult (for users and us) to avoid baking in the rank (dimensionality) of a Tensor into a graph.

- This is because user code commonly branches on the rank of a Tensor (if dim == 2 then do f() else do g()).
- PyTorch framework code also does this: e.g. the implementation of matmul depends on the rank of the Tensor.
- Rank generalized code is also more difficult to symbolically reason over, as you must reason about (unknown length) lists rather than equations over a fixed number of unknown integers.

Furthermore, there is little benefit to avoiding rank-specialization: the claim is that most Pytorch programs will not use inputs with a large number of different ranks. All of the proposals mentioned share this in common: baking in ranks to the graph is OK.

Actionable: We've been saying "rank-specialization is OK" but we don't have empirical evidence. A data-driven analysis of all of the shapes used in e.g. the torchbenchmark suite would help here.

Symbolic Shapes (in Python...)

torch.fx (unlike LazyTensor/AOTAutograd) is able to extract dynamic-shape traces.

- When one calls Tensor.shape, torch.fx returns a “Proxy” object.
- The Proxy object records all operations that happen to it in the fx graph.
- So, in the flatten example, y.shape returns a Proxy object, y.shape[0] is another Proxy, numel is the product of two Proxys, and the indexing, multiplication, and usage in Tensor.view all get recorded!

```
def sin_then_flatten(x):
  y = x.sin()
  numel = y.shape[0] * y.shape[1]
  return y.view(numel)
```

On the LazyTensor side, Nick Korovaiko is prototyping something similar to be used with torch/xla. LazyTensor.shape returns a “DynamicShape” object.

How to handle C++?

It’s important to note that **the previous solution only works in Python**. In C++, Tensor.sizes() is statically typed to return an IntArrayRef. Figuring out something that works in C++ is a big part of the problem.

From first principles, we could change Tensor.sizes() to return “Tuple[Union[Int, SymbolicInt]]”. However, this leads to some performance and memory concerns:

- A Union[Int, SymbolicInt] might be more than 64 bits (uh oh)
- All of PyTorch C++ code would need to be updated to understand that sizes are Union[Int, SymbolicInt]...

Actionable: We don’t have a clear picture of how different C++ solutions would slow down the fast-path (standard eager PyTorch). Ideally we would measure this somehow and use it to determine how aggressive of a design we need.

How to handle C++?

Integer packing **(the leading proposal)**

See https://docs.google.com/document/d/1iiLNwR5ohAsw_ymfnOpDsyF6L9RTUaHMpD8YLw-jxEw/edit for the winning proposal!

To avoid performance implications... we can do some bit-packing.

We can have a simple enum (let's call it SizeVal in this discussion) that can be both an int64_t or represent a pointer to an abstract structure. In particular, we know that all values `< -1` are invalid sizes and can be used to store such pointers.

A simple upgrade path to make a function support dynamic size is to replace all explicit mentions of int64_t by SizeVal.

From offline discussions, a simple way to move toward this without having to do a giant PR is to introduce a new `t.dynamic_sizes()` that returns such objects. The functions that need to be updated then just do things as usual.
Using this, we could migrate targeted c++ subsystems to work with dynamic shapes without major changes to code we don't try to get dynamic shapes through.

Older discussions below:
This solves a "representation" problem (aka, how do we represent our dynamic sizes) but does not solve a dispatch problem. For example -- how would c10::multiply_ints know that it is operating on symbolic ints and not a good-to-honest int64_t size?

```
Tensor flatten(const Tensor& x) {
  int64_t numel = c10::multiply_ints(x.sizes().begin(), x.sizes().end());
  return x.view(numel)
}
```

Discussions here

Do it in Python and pray to the transpiler


```
Tensor flatten(const Tensor& x) {
  int64_t numel = c10::multiply_ints(x.sizes().begin(), x.sizes().end());
  return x.view(numel)
}
```


Back to the flatten example - if we could somehow run the above code in Python (instead of in C++) while tracing... then we could use the "Symbolic Shape in Python" solution and not worry about modifying C++.

- This is really easy to hack, e.g. LazyTensor.flatten does something special that runs the above code in Python.
- A more long-term solution is: instead of maintaining a Python version of flatten and a C++ version, we pray for a Python->C++ transpiler and rewrite problematic things in Python

This doesn't answer the question of what to do when subsystems bake in sizes/shapes though -- rewrite many parts of PyTorch in Python and use a transpiler?

There's also user's C++ code (custom ops) that we won't have access to. The user may be willing to rewrite their code in python but we need to make sure that the transpiler tool will be easy to use for users as well.

There could be a performance concern if we have to do substantial rewrites into python from C++

This approach may not compose nicely with other pytorch systems such as functionalization, autograd.
Consider implementing `view` in python. Functionalization kernels intercept calls to `view` at the dispatcher level, set up alias tracking and then dispatch to `view_copy` on a device. If we intercept `view` in python and dispatch it to the lazy implementation that can handle dynamic shapes, we will need to continue maintaining all the view infrastructure since functionalization kernels wouldn't know how to handle it.

Tensor.sizes returns a Tuple[Tensor]

Say that Tensor.sizes returns a Tuple[Tensor]. All shape math would go through the dispatcher. There's also a slight change in the semantics of operations such as addition assignment. Consider this example:

```
>>> a = 2
>>> b = a
>>> b += 2
>>> print(a, b)
2 4

>>> import torch
>>> a = torch.tensor([2])
>>> b = a
>>> b += 2
>>> print(a, b)
tensor([4]) tensor([4])
```

Segmenting graphs on ops that use static shapes and stitching graphs together later

If we don't have dependencies lazy graph 2 depends on lazy graph 1 we could consider stitching two graphs together and making the input to view a parameter

```
lazy graph1 :

   produces x,y

c++:
  d = x.size(0) + y.size(1)


lazy graph 2:
   z.view(d)
```

Actionable: Identify the cases in either Bert or MaskRCNN where this approach may work. Do we think these cases are prevalent enough. Ponder on the feasibility of graph stitching.

Other directions

Mintorch

A lot of these are research-y and need more investigation but I think they are promising because they aim to solve the problem at the root and that we should seriously consider bringing them to PyTorch.

Zach's Mintorch project has a concept of first-class dimension objects. These are local variables that bind to dimensions of a Tensor and can be manipulated and passed to PyTorch operations. For example, in mintorch, a user could rewrite the flatten example as (pseudocode):

```
def flatten(x):
  i, j = dim()
  x = match(x, [i, j])
  # reshape combines dimensions i and j
  return x.reshape(i, j)
```

And this would, in theory, produce shape-independent traces under the assumption that dimension object manipulation goes through the PyTorch dispatcher. The best thing about it is that **assertions can also be captured in the graph**! The match statement has an implicit assertion that x has two dimensions.

Dex (from Google) also has a notion of dimension indices and JAX is developing a type system based off of that.

Falling back to TS (TorchDynamo) to capture

The issue of capturing symbolic sizes when tracing seems to be somewhat similar to the issue of capturing CF when running torch.jit.trace. Our recommendation to our users was to outline problematic code into a function and `torch.jit.script` it. LTC could trace such cases by creating a JitFunctionCall node which eventually will be inlined into a lowered graph.
This approach might be able to handle a number of hard-to-trace scenarios such as CF, size arithmetic. Unfortunately, it would require a user to refactor model code or it would require us to automate refactoring for the user.
Also, this approach won't work with the XLA backend as they aren't using the TS integration point.